

Query-Subquery Nets

Linh Anh Nguyen¹ and Son Thanh Cao²

¹ Institute of Informatics, University of Warsaw
Banacha 2, 02-097 Warsaw, Poland

nguyen@mimuw.edu.pl

² Faculty of Information Technology, Vinh University
182 Le Duan street, Vinh, Nghe An, Vietnam
sonct@vinhuni.edu.vn

Abstract. We formulate query-subquery nets and use them to create the first framework for developing algorithms for evaluating queries to Horn knowledge bases with the properties that: the approach is goal-directed; each subquery is processed only once and each supplement tuple, if desired, is transferred only once; operations are done set-at-a-time; and any control strategy can be used. Our intention is to increase efficiency of query processing by eliminating redundant computation, increasing flexibility and reducing the number of accesses to the secondary storage. The framework forms a generic evaluation method called QSQN. To deal with function symbols, we use a term-depth bound for atoms and substitutions occurring in the computation and propose to use iterative deepening search which iteratively increases the term-depth bound. We prove soundness and completeness of our generic evaluation method and show that, when the term-depth bound is fixed, the method has PTIME data complexity. We also present how tail recursion elimination can be incorporated into our framework and propose two exemplary control strategies, one is to reduce the number of accesses to the secondary storage, while the other is depth-first search.

Keywords: query processing, Datalog, Horn knowledge bases, QSQ, QSQR, QSQN, QSQTRE, magic-set transformation

1 Introduction

Horn knowledge bases are definite logic programs, which are usually so big that either they cannot be totally loaded into the computer memory or evaluations for them cannot be done totally in the computer memory. Thus, in contrast to logic programming, for Horn knowledge bases efficient access to the secondary storage is an important aspect. Horn knowledge bases can be treated as extensions of Datalog deductive databases without the range-restrictedness and function-free conditions.

This work studies query processing for Horn knowledge bases. It is a continuation of Madalińska-Bugaj and Nguyen's work [7]. As argued in [7], the Horn fragment of first-order logic plays an important role in knowledge representation and reasoning. The QSQN (query-subquery net) evaluation method provided in the current paper is essentially different from the QSQR (query-subquery recursive) method of [7]. However, some introductory and preliminary texts are borrowed from [7].

An efficient method for evaluating queries to Horn knowledge bases should:

- be goal-directed, i.e. the computation should be closely related to the given goal
- be set-oriented (instead of tuple-oriented) in order to reduce the number of accesses to the secondary storage
- do no redundant computation (or do it as less as possible).

As discussed in [7], to develop evaluation procedures for Horn knowledge bases one can either adapt tabled SLD-resolution systems of logic programming to reduce the number of accesses to the secondary storage or generalize evaluation methods of Datalog to deal with non-range-restricted definite logic programs and goals that may contain function symbols.

Tabled SLD-resolution systems like OLDT [17], SLD-AL [19,20], linear tabulated resolution [14,21] are efficient computational procedures for logic programming without redundant recomputations, but they are not directly applicable to Horn knowledge bases to obtain efficient evaluation engines because they are not set-oriented (set-at-a-time). In particular, the suspension-resumption mechanism and the stack-wise representation as well as the “global optimizations of SLD-AL” are all tuple-oriented (tuple-at-a-time). Data structures for them are too complex so that they must be dropped if one wants to convert the methods to efficient set-oriented ones. The try of converting XSB [12,13] (a state-of-the-art implementation of OLDT) to Breadth-First XSB [5] as a set-oriented engine [5] for Horn knowledge bases removes essential features of XSB. Besides, as shown in Example 1.1, the breadth-first approach is not always efficient.

As well-known evaluation methods for Datalog deductive databases, there are the top-down methods QSQR [18], QoSaq [20], QSQ [18,20,1] and the bottom-up method based on magic-set transformation and seminaive evaluation [3,10,1]. As the QSQ approach (including QSQR and QoSaq) is based on SLD-resolution and the magic-set technique simulates QSQ, all of the mentioned methods are goal-directed.

The first version of the QSQR (query-subquery recursive) evaluation method was formulated by Vieille in [18] for Datalog deductive databases. It is set-oriented and uses a tabulation technique. That version is incomplete [20,8]. As pointed out by Mohamed Yahya [7], the version given in the book [1] by Abiteboul et al. is also incomplete. In [7], Madalińska-Bugaj and Nguyen corrected and generalized the method for Horn knowledge bases. The correction depends on clearing global *input* relations for each iteration of the main loop. As observed by Vieille [20], the QSQR approach is like iterative deepening search. It allows redundant recomputations (see [7, Remark 3.2]).

The QoSaq evaluation method [20] is Vieille’s adaptation of SLD-AL resolution for Datalog deductive databases. This evaluation method can be implemented as a set-oriented procedure, but Vieille stated that “*We would like, however, to go even further and to claim that the practical interest of our approach lies in its one-inference-at-a-time basis, as opposed to having a set-theoretic basis. First, this tuple-based computational model permits a fine analysis of the duplicate elimination issue. . . .*” [20, page 5]. Moreover, the specific techniques of QoSaq like “instantiation pattern”, “rule compilation”, “projection” are heavily based on the range-restrictedness and function-free conditions.

The magic-set technique [3,10] for Datalog deductive databases simulates the top-down QSQR evaluation by rewriting a given query to another equivalent one that when evaluated using a bottom-up technique (e.g. the seminaive evaluation) produces only facts produced by the QSQR evaluation. Some authors have extended the magic-set technique for Horn knowledge bases [9,5]. The bottom-up techniques usually use breadth-first search, and as shown in Example 1.1, are not always efficient. The magic-set transformation does not help for the case of that example.

Example 1.1. The order of program clauses and the order of atoms in the bodies of program clauses may be essential, e.g., when the positive logic program defining intensional predicates is specified using the Prolog programming style. In such cases, the top-down depth-first approach may be much more efficient than the breadth-first approaches (including the one based on magic-set transformation and bottom-up seminaive evaluation). Here is such an example, in which x, y, z denote variables and $a_i, b_{i,j}$ denote constant symbols:

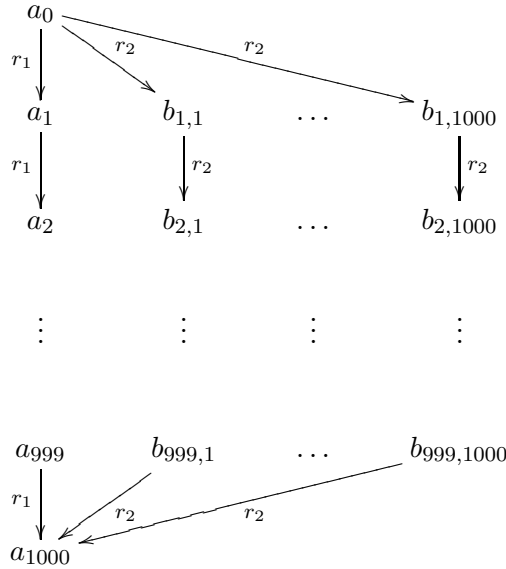
- the positive logic program (for defining intensional predicates p , q_1 and q_2):

$$\begin{aligned}
p &\leftarrow q_1(a_0, a_{1000}) \\
p &\leftarrow q_2(a_0, a_{1000}) \\
q_1(x, y) &\leftarrow r_1(x, y) \\
q_1(x, y) &\leftarrow r_1(x, z), q_1(z, y) \\
q_2(x, y) &\leftarrow r_2(x, y) \\
q_2(x, y) &\leftarrow r_2(x, z), q_2(z, y)
\end{aligned}$$

- the extensional instance (for specifying extensional predicates r_1 and r_2):

$$\begin{aligned}
I(r_1) &= \{(a_i, a_{i+1}) \mid 0 \leq i < 1000\} \\
I(r_2) &= \{(a_0, b_{1,j}) \mid 1 \leq j \leq 1000\} \cup \\
&\quad \{(b_{i,j}, b_{i+1,j}) \mid 1 \leq i < 999 \text{ and } 1 \leq j \leq 1000\} \cup \\
&\quad \{(b_{999,j}, a_{1000}) \mid 1 \leq j \leq 1000\}
\end{aligned}$$

i.e.,



- the goal: $\leftarrow p$.

Our postulate is that the breadth-first approaches (including the evaluation method based on magic-set transformation and bottom-up seminaïve evaluation) are too inflexible and not always efficient. Of course, depth-first search is not always good either. \triangleleft

The QSQ (query-subquery) approach for Datalog queries, as presented in [1], originates from the QSQR method but allows a variety of control strategies. The QSQ framework [18,1] uses adornments to simulate SLD-resolution in pushing constant symbols from goals to subgoals. The annotated version of QSQ also uses annotations to simulate SLD-resolution in pushing repeats of variables from goals to subgoals (see [1]).

In this paper we generalize the QSQ approach for Horn knowledge bases. We formulate query-subquery nets and use them to create the first framework for developing algorithms for evaluating queries to Horn knowledge bases with the following properties:

- the approach is goal-directed

- each subquery is processed only once
- each supplement tuple, if desired, is transferred only once
- operations are done set-at-a-time
- any control strategy can be used.

Our intention is to increase efficiency of query processing by eliminating redundant computation, increasing flexibility and reducing the number of accesses to the secondary storage. The framework forms a generic evaluation method called QSQN. Similarly to [7] but in contrast to the QSQ framework for Datalog queries [1], it does not use adornments and annotations (but has the effects of the annotated version). To deal with function symbols, we use a term-depth bound for atoms and substitutions occurring in the computation and propose to use iterative deepening search which iteratively increases the term-depth bound. We prove soundness and completeness of our generic evaluation method and show that, when the term-depth bound is fixed, the method has PTIME data complexity. We also present how tail recursion elimination [11] can be incorporated into our framework and propose two exemplary control strategies, one is to reduce the number of accesses to the secondary storage, while the other is depth-first search.

The rest of this paper is structured as follows. In Section 2 we recall some notions of first-order logic, logic programming, and Horn knowledge bases. In Section 3 we present our QSQN evaluation method for Horn knowledge bases. We prove its soundness and completeness in Section 4 and estimate its data complexity in Section 5. We consider tail recursion elimination in Section 6 and propose exemplary control strategies for our method in Section 7. Concluding remarks are given in Section 8.

2 Preliminaries

First-order logic is considered in this work and we assume that the reader is familiar with it. We recall only the most important definitions for our work and refer the reader to [6,2] for further reading.

A signature for first-order logic consists of constant symbols, function symbols, and predicate symbols. Terms and formulas over a fixed signature are defined using the symbols of the signature and variables in the usual way. An *atom* is a formula of the form $p(t_1, \dots, t_n)$, where p is an n -ary predicate and t_1, \dots, t_n are terms. An *expression* is either a term, a tuple of terms, a formula without quantifiers or a list of formulas without quantifiers. A *simple expression* is either a term or an atom. The *term-depth of an expression* is the maximal nesting depth of function symbols occurring in that expression.

2.1 Substitution and Unification

A *substitution* is a finite set $\theta = \{x_1/t_1, \dots, x_k/t_k\}$, where x_1, \dots, x_k are pairwise distinct variables, t_1, \dots, t_k are terms, and $t_i \neq x_i$ for all $1 \leq i \leq k$. The set $\text{dom}(\theta) = \{x_1, \dots, x_k\}$ is called the *domain* of θ , while the set $\text{range}(\theta) = \{t_1, \dots, t_k\}$ is called the *range* of θ . By ε we denote the *empty substitution*. The restriction of a substitution θ to a set X of variables is the substitution $\theta|_X = \{(x/t) \in \theta \mid x \in X\}$. The *term-depth of a substitution* is the maximal nesting depth of function symbols occurring in that substitution.

Let $\theta = \{x_1/t_1, \dots, x_k/t_k\}$ be a substitution and E be an expression. Then $E\theta$, the *instance* of E by θ , is the expression obtained from E by simultaneously replacing all occurrences of the variable x_i in E by the term t_i , for $1 \leq i \leq k$.

Let $\theta = \{x_1/t_1, \dots, x_k/t_k\}$ and $\delta = \{y_1/s_1, \dots, y_h/s_h\}$ be substitutions (where x_1, \dots, x_k are pairwise distinct variables, and y_1, \dots, y_h are also pairwise distinct vari-

ables). Then the *composition* $\theta\delta$ of θ and δ is the substitution obtained from the sequence $\{x_1/(t_1\delta), \dots, x_k/(t_k\delta), y_1/s_1, \dots, y_h/s_h\}$ by deleting any binding $x_i/(t_i\delta)$ for which $x_i = (t_i\delta)$ and deleting any binding y_j/s_j for which $y_j \in \{x_1, \dots, x_k\}$.

A substitution θ is *idempotent* if $\theta\theta = \theta$. It is known that $\theta = \{x_1/t_1, \dots, x_k/t_k\}$ is idempotent if none of x_1, \dots, x_k occurs in any t_1, \dots, t_k .

If θ and δ are substitutions such that $\theta\delta = \delta\theta = \varepsilon$, then we call them *renaming substitutions*. We say that an expression E is a *variant* of an expression E' if there exist substitutions θ and γ such that $E = E'\theta$ and $E' = E\gamma$.

A substitution θ is *more general* than a substitution δ if there exists a substitution γ such that $\delta = \theta\gamma$. Note that according to this definition, θ is more general than itself.

Let Γ be a set of simple expressions. A substitution θ is called a *unifier* for Γ if $\Gamma\theta$ is a singleton. If $\Gamma\theta = \{\varphi\}$ then we say that θ unifies Γ (into φ). A unifier θ for Γ is called a *most general unifier* (mgu) for Γ if θ is more general than every unifier of Γ .

There is an effective algorithm, called the *unification algorithm*, for checking whether a set Γ of simple expressions is unifiable (i.e. has a unifier) and computing an idempotent mgu for Γ if Γ is unifiable (see, e.g., [6]).

If E is an expression or a substitution then by $\text{Vars}(E)$ we denote the set of variables occurring in E . If φ is a formula then by $\forall(\varphi)$ we denote the *universal closure* of φ , which is the formula obtained by adding a universal quantifier for every variable having a free occurrence in φ .

2.2 Positive Logic Programs and SLD-Resolution

A (positive or definite) *program clause* is a formula of the form $\forall(A \vee \neg B_1 \vee \dots \vee \neg B_k)$ with $k \geq 0$, written as $A \leftarrow B_1, \dots, B_k$, where A, B_1, \dots, B_k are atoms. A is called the *head*, and (B_1, \dots, B_k) the *body* of the program clause. If p is the predicate of A then the program clause is called a program clause defining p .

A *positive* (or *definite*) *logic program* is a finite set of program clauses.

A *goal* (also called a *negative clause*) is a formula of the form $\forall(\neg B_1 \vee \dots \vee \neg B_k)$, written as $\leftarrow B_1, \dots, B_k$, where B_1, \dots, B_k are atoms. If $k = 1$ then the goal is called a *unary goal*. If $k = 0$ then the goal stands for falsity and is called the *empty goal* (or the *empty clause*) and denoted by \square .

If P is a positive logic program and $G = \leftarrow B_1, \dots, B_k$ is a goal, then θ is called a *correct answer* for $P \cup \{G\}$ if $P \models \forall((B_1 \wedge \dots \wedge B_k)\theta)$.

We now give definitions for SLD-resolution.

A goal G' is *derived* from a goal $G = \leftarrow A_1, \dots, A_i, \dots, A_k$ and a program clause $\varphi = (A \leftarrow B_1, \dots, B_h)$ using A_i as the *selected atom* and θ as the most general unifier (mgu) if θ is an mgu for A_i and A , and $G' = \leftarrow (A_1, \dots, A_{i-1}, B_1, \dots, B_h, A_{i+1}, \dots, A_k)\theta$. We call G' a *resolvent* of G and φ . If $i = 1$ then we say that G' is derived from G and φ using the *leftmost selection function*.

Let P be a positive logic program and G be a goal.

An *SLD-derivation* from $P \cup \{G\}$ consists of a (finite or infinite) sequence $G_0 = G, G_1, G_2, \dots$ of goals, a sequence $\varphi_1, \varphi_2, \dots$ of variants of program clauses of P and a sequence $\theta_1, \theta_2, \dots$ of mgu's such that each G_{i+1} is derived from G_i and φ_{i+1} using θ_{i+1} . Each φ_i is a suitable variant of the corresponding program clause. That is, φ_i does not have any variables which already appear in the derivation up to G_{i-1} . Each program clause variant φ_i is called an *input program clause*.

An *SLD-refutation* of $P \cup \{G\}$ is a finite SLD-derivation from $P \cup \{G\}$ which has the empty clause as the last goal in the derivation.

A *computed answer* θ for $P \cup \{G\}$ is the substitution obtained by restricting the composition $\theta_1 \dots \theta_n$ to the variables of G , where $\theta_1, \dots, \theta_n$ is the sequence of mgu's occurring in an SLD-refutation of $P \cup \{G\}$.

Theorem 2.1 (Soundness and Completeness of SLD-Resolution [4,16]). *Let P be a positive logic program and G be a goal. Then every computed answer for $P \cup \{G\}$ is a correct answer for $P \cup \{G\}$. Conversely, for every correct answer θ for $P \cup \{G\}$, using any selection function there exists a computed answer δ for $P \cup \{G\}$ such that $G\theta = G\delta\gamma$ for some substitution γ .* \triangleleft

We will use also the following well-known lemmas:

Lemma 2.2 (Lifting Lemma). *Let P be a positive logic program, G be a goal, θ be a substitution, and l be a natural number. Suppose there exists an SLD-refutation of $P \cup \{G\theta\}$ using mgu's $\theta_1, \dots, \theta_n$ such that the variables of the input program clauses are distinct from the variables in G and θ and the term-depths of the goals are bounded by l . Then there exist a substitution γ and an SLD-refutation of $P \cup \{G\}$ using the same sequence of input program clauses, the same selected atoms and mgu's $\theta'_1, \dots, \theta'_n$ such that the term-depths of the goals are bounded by l and $\theta\theta_1 \dots \theta_n = \theta'_1 \dots \theta'_n \gamma$.* \triangleleft

The Lifting Lemma given in [6] does not contain the condition “the variables of the input program clauses are distinct from the variables in G and θ ” and is therefore inaccurate (see, e.g., [2]). The correct version given above follows from the one presented, amongst others, in [15]. For applications of this lemma in this paper, we assume that *fresh variables* from a special infinite list of variables are used for renaming variables of input program clauses in SLD-derivations, and that mgu's are computed using a standard method. The mentioned condition will thus be satisfied.

In a computational process, a *fresh variant* of a formula φ , where φ can be an atom, a goal $\leftarrow A$ or a program clause $A \leftarrow B_1, \dots, B_k$ (written without quantifiers), is a formula $\varphi\theta$, where θ is a renaming substitution such that $\text{dom}(\theta) = \text{Vars}(\varphi)$ and $\text{range}(\theta)$ consists of fresh variables that were not used in the computation (and the input).

2.3 Definitions for Horn Knowledge Bases

Similarly as for deductive databases, we classify each predicate either as *intensional* or as *extensional*. A *generalized tuple* is a tuple of terms, which may contain function symbols and variables. A *generalized relation* is a set of generalized tuples of the same arity. A *Horn knowledge base* is defined to be a pair consisting of a positive logic program for defining intensional predicates and a *generalized extensional instance*, which is a function mapping each extensional n -ary predicate to an n -ary generalized relation. Note that intensional predicates are defined by a positive logic program which may contain function symbols and not be range-restricted. From now on, we use the term “relation” to mean a generalized relation, and the term “extensional instance” to mean a generalized extensional instance.

Given a Horn knowledge base specified by a positive logic program P and an extensional instance I , a *query* to the knowledge base is a positive formula $\varphi(\bar{x})$ without quantifiers, where \bar{x} is a tuple of all the variables of φ .³ A (*correct*) *answer* for the query is a tuple \bar{t} of terms of the same length as \bar{x} such that $P \cup I \models \forall(\varphi(\bar{t}))$. When measuring *data complexity*, we assume that P and φ are fixed, while I varies. Thus, the pair $(P, \varphi(\bar{x}))$ is treated as a *query* to the extensional instance I . We will use the term “query” in this meaning.

³ A *positive formula without quantifiers* is a formula built up from atoms using only connectives \wedge and \vee .

It can easily be shown that, every query $(P, \varphi(\bar{x}))$ can be transformed in polynomial time to an equivalent query of the form $(P', q(\bar{x}))$ over a signature extended with new intensional predicates, including q . The equivalence means that, for every extensional instance I and every tuple \bar{t} of terms of the same length as \bar{x} , $P \cup I \models \forall(\varphi(\bar{t}))$ iff $P' \cup I \models \forall(q(\bar{t}))$. The transformation is based on introducing new predicates for defining complex subformulas occurring in the query. For example, if $\varphi = p(x) \wedge r(x, y)$, then $P' = P \cup \{q(x, y) \leftarrow p(x), r(x, y)\}$, where q is a new intensional predicate.

Without loss of generality, we will consider only queries of the form $(P, q(\bar{x}))$, where q is an intensional predicate. Answering such a query on an extensional instance I is to find (correct) answers for $P \cup I \cup \{\leftarrow q(\bar{x})\}$.

3 Query-Subquery Nets

Let P be a positive logic program and $\varphi_1, \dots, \varphi_m$ be all the program clauses of P , with

$$\varphi_i = (A_i \leftarrow B_{i,1}, \dots, B_{i,n_i})$$

where $n_i \geq 0$. A *query-subquery net structure* (in short, *QSQ-net structure*) of P is a tuple (V, E, T) such that:

- V consists of nodes
 - $input_p$ and ans_p for each intensional predicate p of P
 - $pre_filter_i, filter_{i,1}, \dots, filter_{i,n_i}, post_filter_i$ for each $1 \leq i \leq m$
- E consists of edges
 - $(filter_{i,1}, filter_{i,2}), \dots, (filter_{i,n_i-1}, filter_{i,n_i})$ for each $1 \leq i \leq m$
 - $(pre_filter_i, filter_{i,1})$ and $(filter_{i,n_i}, post_filter_i)$ for each $1 \leq i \leq m$ with $n_i \geq 1$
 - $(pre_filter_i, post_filter_i)$ for each $1 \leq i \leq m$ with $n_i = 0$
 - $(input_p, pre_filter_i)$ and $(post_filter_i, ans_p)$ for each $1 \leq i \leq m$, where p is the predicate of A_i
 - $(filter_{i,j}, input_p)$ and $(ans_p, filter_{i,j})$ for each intensional predicate p and each $1 \leq i \leq m$ and $1 \leq j \leq n_i$ such that $B_{i,j}$ is an atom of p
- T is a function, called the *memorizing type* of the net structure, mapping each node $filter_{i,j} \in V$ such that the predicate of $B_{i,j}$ is extensional to *true* or *false*.

If $(v, w) \in E$ then we call w a *successor* of v , and v a *predecessor* of w . Note that V and E are uniquely specified by P . We call the pair (V, E) the *QSQ topological structure* of P .

Example 3.1. Figure 1 illustrates the QSQ topological structure of the following positive logic program:

$$\begin{aligned} p(x, y) &\leftarrow q(x, y) \\ p(x, y) &\leftarrow q(x, z), p(z, y). \end{aligned}$$

◁

A *query-subquery net* (in short, *QSQ-net*) of P is a tuple $N = (V, E, T, C)$ such that (V, E, T) is a QSQ-net structure of P and C is a mapping that associates each node $v \in V$ with a structure called the *contents* of v , satisfying the following conditions:

- $C(v)$, where $v = input_p$ or $v = ans_p$ for an intensional predicate p of P , consists of:
 - $tuples(v)$: a set of generalized tuples of the same arity as p
 - $unprocessed(v, w)$ for $(v, w) \in E$: a subset of $tuples(v)$
- $C(v)$, where $v = pre_filter_i$, consists of:
 - $atom(v) = A_i$ and $post_vars(v) = Vars((B_{i,1}, \dots, B_{i,n_i}))$

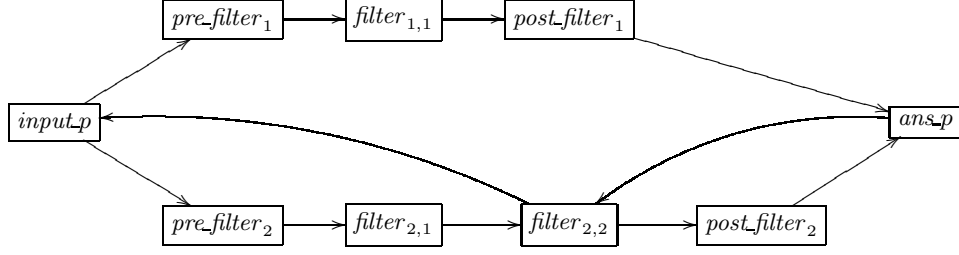


Fig. 1. The QSQ topological structure of the program given in Example 3.1.

- $C(v)$, where $v = \text{post_filter}_i$, is empty, but we assume $\text{pre_vars}(v) = \emptyset$
- $C(v)$, where $v = \text{filter}_{i,j}$ and p is the predicate of $B_{i,j}$, consists of:
 - $\text{kind}(v) = \text{extensional}$ if p is extensional, and $\text{kind}(v) = \text{intensional}$ otherwise
 - $\text{pred}(v) = p$ and $\text{atom}(v) = B_{i,j}$
 - $\text{pre_vars}(v) = \text{Vars}((B_{i,j}, \dots, B_{i,n_i}))$ and $\text{post_vars}(v) = \text{Vars}((B_{i,j+1}, \dots, B_{i,n_i}))$
 - $\text{subqueries}(v)$: a set of pairs of the form (\bar{t}, δ) , where \bar{t} is a generalized tuple of the same arity as the predicate of A_i and δ is an idempotent substitution such that $\text{dom}(\delta) \subseteq \text{pre_vars}(v)$ and $\text{dom}(\delta) \cap \text{Vars}(\bar{t}) = \emptyset$
 - $\text{unprocessed_subqueries}(v) \subseteq \text{subqueries}(v)$
 - in the case p is intensional:
 - * $\text{unprocessed_subqueries}_2(v) \subseteq \text{subqueries}(v)$
 - * $\text{unprocessed_tuples}(v)$: a set of generalized tuples of the same arity as p
- if $v = \text{filter}_{i,j}$, $\text{kind}(v) = \text{extensional}$ and $T(v) = \text{false}$ then $\text{subqueries}(v) = \emptyset$.

By a *subquery* we mean a pair of the form (\bar{t}, δ) , where \bar{t} is a generalized tuple and δ is an idempotent substitution such that $\text{dom}(\delta) \cap \text{Vars}(\bar{t}) = \emptyset$.

For $v = \text{filter}_{i,j}$ and p being the predicate of A_i , the meaning of a subquery $(\bar{t}, \delta) \in \text{subqueries}(v)$ is that: for processing the goal $\leftarrow p(\bar{s})$ with $\bar{s} \in \text{tuples}(\text{input}_p)$ using the program clause $\varphi_i = (A_i \leftarrow B_{i,1}, \dots, B_{i,n_i})$, unification of $p(\bar{s})$ and A_i as well as processing of the subgoals $B_{i,1}, \dots, B_{i,j-1}$ were done, amongst others, by using a sequence of mgu's $\gamma_0, \dots, \gamma_{j-1}$ with the property that $\bar{t} = \bar{s}\gamma_0 \dots \gamma_{j-1}$ and $\delta = (\gamma_0 \dots \gamma_{j-1})|_{\text{Vars}((B_{i,j}, \dots, B_{i,n_i}))}$.

An *empty QSQ-net* of P is a QSQ-net of P with all the sets of the forms $\text{tuples}(v)$, $\text{unprocessed}(v, w)$, $\text{subqueries}(v)$, $\text{unprocessed_subqueries}(v)$, $\text{unprocessed_subqueries}_2(v)$, $\text{unprocessed_tuples}(v)$ being empty.

In a QSQ-net, if $v = \text{pre_filter}_i$ or $v = \text{post_filter}_i$ or $v = \text{filter}_{i,j}$ and $\text{kind}(v) = \text{extensional}$ then v has exactly one successor, which we denote by $\text{succ}(v)$.

If v is $\text{filter}_{i,j}$ with $\text{kind}(v) = \text{intensional}$ and $\text{pred}(v) = p$ then v has exactly two successors. In that case, let

$$\text{succ}(v) = \begin{cases} \text{filter}_{i,j+1} & \text{if } n_i > j \\ \text{post_filter}_i & \text{otherwise} \end{cases}$$

and $\text{succ}_2(v) = \text{input}_p$. The set $\text{unprocessed_subqueries}(v)$ is used for (i.e. corresponds to) the edge $(v, \text{succ}(v))$, while $\text{unprocessed_subqueries}_2(v)$ is used for the edge $(v, \text{succ}_2(v))$.

Note that if $\text{succ}(v) = w$ then $\text{post_vars}(v) = \text{pre_vars}(w)$. In particular, $\text{post_vars}(\text{filter}_{i,n_i}) = \text{pre_vars}(\text{post_filter}_i) = \emptyset$.

The formats of data transferred through edges of a QSQ-net are specified as follows:

- data transferred through an edge of the form (input_p, v) , (v, input_p) , (v, ans_p) or (ans_p, v) is a finite set of generalized tuples of the same arity as p

- data transferred through an edge (u, v) with $v = filter_{i,j}$ and u not being of the form ans_p is a finite set of subqueries that can be added to $subqueries(v)$
- data transferred through an edge $(v, post_filter_i)$ is a set of subqueries (\bar{t}, ε) such that \bar{t} is a generalized tuple of the same arity as the predicate of A_i .

If (\bar{t}, δ) and (\bar{t}', δ') are subqueries that can be transferred through an edge to v then we say that (\bar{t}, δ) is *more general* than (\bar{t}', δ') w.r.t. v , and that (\bar{t}', δ') is *less general* than (\bar{t}, δ) w.r.t. v , if there exists a substitution γ such that $\bar{t}\gamma = \bar{t}'$ and $(\delta\gamma)|_{pre_vars(v)} = \delta'$.

Informally, a subquery (\bar{t}, δ) transferred through an edge to v is processed as follows:

- if $v = filter_{i,j}$, $kind(v) = extensional$ and $pred(v) = p$ then, for each $\bar{t}' \in I(p)$, if $atom(v)\delta = B_{i,j}\delta$ is unifiable with a fresh variant of $p(\bar{t}')$ by an mgu γ then transfer the subquery $(\bar{t}\gamma, (\delta\gamma)|_{post_vars(v)})$ through $(v, succ(v))$
- if $v = filter_{i,j}$, $kind(v) = intensional$ and $pred(v) = p$ then
 - transfer the input tuple \bar{t}' such that $p(\bar{t}') = atom(v)\delta = B_{i,j}\delta$ through $(v, input_p)$ to add a fresh variant of it to $tuples(input_p)$
 - for each currently existing $\bar{t}' \in tuples(ans_p)$, if $atom(v)\delta = B_{i,j}\delta$ is unifiable with a fresh variant of $p(\bar{t}')$ by an mgu γ then transfer the subquery $(\bar{t}\gamma, (\delta\gamma)|_{post_vars(v)})$ through $(v, succ(v))$
 - store the subquery (\bar{t}, δ) in $subqueries(v)$, and later, for each new \bar{t}' added to $tuples(ans_p)$, if $atom(v)\delta = B_{i,j}\delta$ is unifiable with a fresh variant of $p(\bar{t}')$ by an mgu γ then transfer the subquery $(\bar{t}\gamma, (\delta\gamma)|_{post_vars(v)})$ through $(v, succ(v))$
- if $v = post_filter_i$ and p is the predicate of A_i then transfer the answer tuple \bar{t} through $(postFilter_i, ans_p)$ to add it to $tuples(ans_p)$.

Formally, the processing of a subquery is designed more sophisticatedly so that:

- every subquery / input tuple / answer tuple subsumed by another one is ignored
- every subquery / input tuple / answer tuple with term-depth greater than l is ignored
- the processing is divided into smaller steps which can be delayed to maximize flexibility and allow various control strategies
- the processing is done set-at-a-time (e.g., for all the unprocessed subqueries accumulated in a given node).

Procedure **transfer** (D, u, v) (given on page 10) specifies the effects of transferring data D through an edge (u, v) of a QSQ-net. If v is of the form pre_filter_i or $post_filter_i$ or $(v = filter_{i,j}$ and $kind(v) = extensional$ and $T(v) = false$) then the input D for v is processed immediately and appropriate data Γ is produced and transferred through $(v, succ(v))$. Otherwise, the input D for v is not processed immediately, but accumulated into the structure of v in an appropriate way.

Function **active-edge** (u, v) (given on page 12) returns *true* for an edge (u, v) if data accumulated in u can be processed to produce some data to transfer through (u, v) , and returns *false* otherwise.

In the case **active-edge** (u, v) is true, procedure **fire** (u, v) (given on page 12) processes data accumulated in u that has not been processed before to transfer appropriate data through the edge (u, v) .

Algorithm 1 (given on page 13) presents our QSQN evaluation method for Horn knowledge bases.

3.1 Relaxing Term-Depth Bound

Suppose that we want to compute as many as possible but no more than k correct answers for a query $(P, q(\bar{x}))$ on an extensional instance I within time limit L . Then we can use

Procedure $\text{transfer}(D, u, v)$

Global data: a Horn knowledge base (P, I) , a QSQ-net $N = (V, E, T, C)$ of P , and a term-depth bound l .

Input: data D to transfer through the edge $(u, v) \in E$.

```

1  if  $D = \emptyset$  then return;
2  if  $u$  is inputp then
3       $\Gamma := \emptyset$ ;
4      foreach  $\bar{t} \in D$  do
5          if  $p(\bar{t})$  and  $\text{atom}(v)$  are unifiable by an mgu  $\gamma$  then
6              add-subquery( $\bar{t}\gamma, \gamma|_{\text{post\_vars}(v)}, \Gamma, \text{succ}(v)$ )
7      transfer( $\Gamma, v, \text{succ}(v)$ )
8  else if  $u$  is ansp then  $\text{unprocessed\_tuples}(v) := \text{unprocessed\_tuples}(v) \cup D$ 
9  else if  $v$  is inputp or ansp then
10     foreach  $\bar{t} \in D$  do
11         let  $\bar{t}'$  be a fresh variant of  $\bar{t}$ ;
12         if  $\bar{t}'$  is not an instance of any tuple from  $\text{tuples}(v)$  then
13             foreach  $\bar{t}'' \in \text{tuples}(v)$  do
14                 if  $\bar{t}''$  is an instance of  $\bar{t}'$  then
15                     delete  $\bar{t}''$  from  $\text{tuples}(v)$ ;
16                     foreach  $(v, w) \in E$  do delete  $\bar{t}''$  from  $\text{unprocessed}(v, w)$ 
17             if  $v$  is inputp then
18                 add  $\bar{t}'$  to  $\text{tuples}(v)$ ;
19                 foreach  $(v, w) \in E$  do add  $\bar{t}'$  to  $\text{unprocessed}(v, w)$ 
20             else
21                 add  $\bar{t}$  to  $\text{tuples}(v)$ ;
22                 foreach  $(v, w) \in E$  do add  $\bar{t}$  to  $\text{unprocessed}(v, w)$ 
23  else if  $v$  is filteri,j and  $\text{kind}(v) = \text{extensional}$  and  $T(v) = \text{false}$  then
24     let  $p = \text{pred}(v)$  and set  $\Gamma := \emptyset$ ;
25     foreach  $(\bar{t}, \delta) \in D$  do
26         if  $\text{term-depth}(\text{atom}(v)\delta) \leq l$  then
27             foreach  $\bar{t}' \in I(p)$  do
28                 if  $\text{atom}(v)\delta$  is unifiable with a fresh variant of  $p(\bar{t}')$  by an mgu  $\gamma$  then
29                     add-subquery( $\bar{t}'\gamma, (\delta\gamma)|_{\text{post\_vars}(v)}, \Gamma, \text{succ}(v)$ )
30     transfer( $\Gamma, v, \text{succ}(v)$ )
31  else if  $v$  is filteri,j and  $(\text{kind}(v) = \text{extensional}$  and  $T(v) = \text{true}$  or  $\text{kind}(v) = \text{intensional})$  then
32     foreach  $(\bar{t}, \delta) \in D$  do
33         if  $\text{term-depth}(\text{atom}(v)\delta) \leq l$  then
34             if no subquery in  $\text{subqueries}(v)$  is more general than  $(\bar{t}, \delta)$  then
35                 delete from  $\text{subqueries}(v)$  all subqueries less general than  $(\bar{t}, \delta)$ ;
36                 delete from  $\text{unprocessed\_subqueries}(v)$  all subqueries less general than  $(\bar{t}, \delta)$ ;
37                 add  $(\bar{t}, \delta)$  to both  $\text{subqueries}(v)$  and  $\text{unprocessed\_subqueries}(v)$ ;
38                 if  $\text{kind}(v) = \text{intensional}$  then
39                     delete from  $\text{unprocessed\_subqueries}_2(v)$  all subqueries less general than  $(\bar{t}, \delta)$ ;
40                     add  $(\bar{t}, \delta)$  to  $\text{unprocessed\_subqueries}_2(v)$ 
41  else //  $v$  is of the form post_filteri
42      $\Gamma := \{\bar{t} \mid (\bar{t}, \varepsilon) \in D\}$ ;
43     transfer( $\Gamma, v, \text{succ}(v)$ )

```

Procedure add-subquery($\bar{t}, \delta, \Gamma, v$)**Purpose:** add the subquery (\bar{t}, δ) to Γ , but keep in Γ only the most general subqueries w.r.t. v .

- 1 **if** $\text{term-depth}(\bar{t}) \leq l$ **and** $\text{term-depth}(\delta) \leq l$ **and** no subquery in Γ is more general than (\bar{t}, δ) w.r.t. v **then**
- 2 delete from Γ all subqueries less general than (\bar{t}, δ) w.r.t. v ;
- 3 add (\bar{t}, δ) to Γ

Procedure add-tuple(\bar{t}, Γ)**Purpose:** add the tuple \bar{t} to Γ , but keep in Γ only the most general tuples.

- 1 let \bar{t}' be a fresh variant of \bar{t} ;
- 2 **if** \bar{t}' is not an instance of any tuple from Γ **then**
- 3 delete from Γ all tuples that are instances of \bar{t}' ;
- 4 add \bar{t}' to Γ

iterative deepening search which iteratively increases term-depth bound for atoms and substitutions occurring in the computation as follows:

1. Initialize term-depth bound l to 0 (or another small natural number).
2. Run Algorithm 1 for evaluating $(P, q(\bar{x}))$ on I within the time limit.
3. While $\text{tuples}(\text{ans}_q)$ contains less than k tuples and the time limit was not reached yet, do:
 - (a) Clear (empty) all the sets of the form $\text{tuples}(\text{input}_p)$ and $\text{subqueries}(\text{filter}_{i,j})$.
 - (b) Increase term-depth bound l by 1.
 - (c) Run Algorithm 1 without Steps 1 and 2.
4. Return $\text{tuples}(\text{ans}_q)$.

4 Soundness and Completeness

The following lemma states a property of Algorithm 1. Its proof is straightforward.

Lemma 4.1. *Consider a run of Algorithm 1 (using parameter l) on a query $(P, q(\bar{x}))$ and an extensional instance I and let (V, E, T, C) be the resulting QSQ-net. Let $v = \text{filter}_{i,j}$ for some $1 \leq i \leq m$ and $1 \leq j \leq n_i$. Let $w = \text{succ}(v)$ and let $u = \text{filter}_{i,j-1}$ if $j > 1$, and $u = \text{pre_filter}_i$ otherwise. Suppose that a subquery (\bar{s}', δ') was transferred through (v, w) at some step k . Then a subquery (\bar{s}, δ) was transferred through (u, v) at some earlier step $h < k$ with the property that:*

- if $\text{kind}(v) = \text{extensional}$ and $\text{pred}(v) = p$ then there exists $\bar{t}' \in I(p)$ such that $\text{atom}(v)\delta$ is unifiable with a fresh variant of $p(\bar{t}')$ by an mgu γ , $\bar{s}' = \bar{s}\gamma$ and $\delta' = (\delta\gamma)_{|\text{post_vars}(v)}$
- if $\text{kind}(v) = \text{intensional}$ and $\text{pred}(v) = p$ then there was $\bar{t}' \in \text{tuples}(\text{ans}_p)$ at step k such that $\text{atom}(v)\delta$ is unifiable with a fresh variant of $p(\bar{t}')$ by an mgu γ , $\bar{s}' = \bar{s}\gamma$ and $\delta' = (\delta\gamma)_{|\text{post_vars}(v)}$. \triangleleft

Theorem 4.2 (Soundness). *After a run of Algorithm 1 on a query $(P, q(\bar{x}))$ and an extensional instance I , for all intensional predicates p of P , every computed answer $\bar{t} \in \text{tuples}(\text{ans}_p)$ is a correct answer in the sense that $P \cup I \models \forall(p(\bar{t}))$.*

Proof. We prove $P \cup I \models \forall(p(\bar{t}))$ by induction on the number of the step at which \bar{t} was added to $\text{tuples}(\text{ans}_p)$. Suppose \bar{t} was added to $\text{tuples}(\text{ans}_p)$ as the result of transferring \bar{t} through the edge $(\text{post_filter}_i, \text{ans}_p)$, which was triggered by the transfer of (\bar{t}, ε) through

Function $\text{active-edge}(u, v)$

Global data: a QSQ-net $N = (V, E, T, C)$.**Input:** an edge $(u, v) \in E$.**Output:** *true* if there are data to transfer through the edge (u, v) , and *false* otherwise.

```

1 if  $u$  is  $\text{pre\_filter}_i$  or  $\text{post\_filter}_i$  then return false
2 else if  $u$  is  $\text{input}_p$  or  $\text{ans}_p$  then return  $\text{unprocessed}(u, v) \neq \emptyset$ 
3 else if  $u$  is  $\text{filter}_{i,j}$  and  $\text{kind}(u) = \text{extensional}$  then
4   return  $T(u) = \text{true} \wedge \text{unprocessed\_subqueries}(u) \neq \emptyset$ 
5 else //  $u$  is of the form  $\text{filter}_{i,j}$  and  $\text{kind}(u) = \text{intensional}$ 
6   let  $p = \text{pred}(u)$ ;
7   if  $v = \text{input}_p$  then return  $\text{unprocessed\_subqueries}_2(u) \neq \emptyset$ 
8   else return  $\text{unprocessed\_subqueries}(u) \neq \emptyset \vee \text{unprocessed\_tuples}(u) \neq \emptyset$ 

```

Procedure $\text{fire}(u, v)$

Global data: a Horn knowledge base (P, I) , a QSQ-net $N = (V, E, T, C)$ of P , and a term-depth bound l .**Input:** an edge $(u, v) \in E$ such that $\text{active-edge}(u, v)$ holds.

```

1 if  $u$  is  $\text{input}_p$  or  $\text{ans}_p$  then
2   transfer( $\text{unprocessed}(u, v), u, v$ );
3    $\text{unprocessed}(u, v) := \emptyset$ 
4 else if  $u$  is  $\text{filter}_{i,j}$  and  $\text{kind}(u) = \text{extensional}$  and  $T(u) = \text{true}$  then
5   let  $p = \text{pred}(u)$  and set  $\Gamma := \emptyset$ ;
6   foreach  $(\bar{t}, \delta) \in \text{unprocessed\_subqueries}(u)$  do
7     foreach  $\bar{t}' \in I(p)$  do
8       if  $\text{atom}(u)\delta$  is unifiable with a fresh variant of  $p(\bar{t}')$  by an mgu  $\gamma$  then
9         add-subquery( $\bar{t}'\gamma, (\delta\gamma)_{|\text{post\_vars}(u)}, \Gamma, v$ )
10    $\text{unprocessed\_subqueries}(u) := \emptyset$ ;
11   transfer( $\Gamma, u, v$ )
12 else if  $u$  is  $\text{filter}_{i,j}$  and  $\text{kind}(u) = \text{intensional}$  then
13   let  $p = \text{pred}(u)$  and set  $\Gamma := \emptyset$ ;
14   if  $v = \text{input}_p$  then
15     foreach  $(\bar{t}, \delta) \in \text{unprocessed\_subqueries}_2(u)$  do let  $p(\bar{t}') = \text{atom}(u)\delta$ , add-tuple( $\bar{t}', \Gamma$ );
16      $\text{unprocessed\_subqueries}_2(u) := \emptyset$ ;
17   else
18     foreach  $(\bar{t}, \delta) \in \text{unprocessed\_subqueries}(u)$  do
19       foreach  $\bar{t}' \in \text{tuples}(\text{ans}_p)$  do
20         if  $\text{atom}(u)\delta$  is unifiable with a fresh variant of  $p(\bar{t}')$  by an mgu  $\gamma$  then
21           add-subquery( $\bar{t}'\gamma, (\delta\gamma)_{|\text{post\_vars}(u)}, \Gamma, v$ )
22      $\text{unprocessed\_subqueries}(u) := \emptyset$ ;
23     if  $\text{unprocessed\_tuples}(u) \neq \emptyset$  then
24       foreach  $\bar{t} \in \text{unprocessed\_tuples}(u)$  do
25         foreach  $(\bar{t}', \delta) \in \text{subqueries}(u)$  do
26           if  $\text{atom}(u)\delta$  is unifiable with a fresh variant of  $p(\bar{t}')$  by an mgu  $\gamma$  then
27             add-subquery( $\bar{t}'\gamma, (\delta\gamma)_{|\text{post\_vars}(u)}, \Gamma, v$ )
28      $\text{unprocessed\_tuples}(u) := \emptyset$ 
29   transfer( $\Gamma, u, v$ )

```

Algorithm 1: for evaluating a query $(P, q(\bar{x}))$ on an extensional instance I .

```

1 let  $(V, E, T)$  be a QSQ-net structure of  $P$ ; //  $T$  can be chosen arbitrarily
2 set  $C$  so that  $N = (V, E, T, C)$  is an empty QSQ-net of  $P$ ;
3 let  $\bar{x}'$  be a fresh variant of  $\bar{x}$ ;
4  $tuples(input\_q) := \{\bar{x}'\}$ ;
5 foreach  $(input\_q, v) \in E$  do  $unprocessed(input\_q, v) := \{\bar{x}'\}$ ;
6 while there exists  $(u, v) \in E$  such that active-edge $(u, v)$  holds do
7    $\left[ \begin{array}{l} \text{select } (u, v) \in E \text{ such that } \mathbf{active-edge}(u, v) \text{ holds;} \\ // \text{ any strategy is acceptable for the above selection} \end{array} \right.$ 
8    $\left[ \begin{array}{l} \text{fire}(u, v) \end{array} \right.$ 
9 return  $tuples(ans\_q)$ 

```

the edge $(filter_{i,n_i}, post_filter_i)$. Let $\bar{s}_{n_i} = \bar{t}$ and $\delta_{n_i} = \varepsilon$. Let $v_0 = pre_filter_i$ and $v_j = filter_{i,j}$ for $1 \leq j \leq n_i$. By Lemma 4.1, for each j from n_i to 1, there exists a subquery $(\bar{s}_{j-1}, \delta_{j-1})$ transferred through (v_{j-1}, v_j) such that:

if $kind(v_j) = extensional$ and $pred(v_j) = p_j$ then there exists $\bar{t}'_j \in I(p_j)$ such that $atom(v_j)\delta_{j-1}$ is unifiable with a fresh variant of $p_j(\bar{t}'_j)$ by an mgu γ_j ,
 $\bar{s}_j = \bar{s}_{j-1}\gamma_j$ and $\delta_j = (\delta_{j-1}\gamma_j)_{|post_vars(v_j)}$ (1)

if $kind(v_j) = intensional$ and $pred(v_j) = p_j$ then there exists $\bar{t}'_j \in tuples(ans_p_j)$ such that $atom(v_j)\delta_{j-1}$ is unifiable with a fresh variant of $p_j(\bar{t}'_j)$ by an mgu γ_j , $\bar{s}_j = \bar{s}_{j-1}\gamma_j$ and $\delta_j = (\delta_{j-1}\gamma_j)_{|post_vars(v_j)}$. (2)

We have that $A_i\delta_0 = p(\bar{s}_0)$. We prove by an inner induction on $1 \leq j \leq n_i + 1$ that:

for every substitution θ , if $P \cup I \models \forall((B_{i,j} \wedge \dots \wedge B_{i,n_i})\delta_{j-1}\theta)$ then
 $P \cup I \models \forall(p(\bar{s}_{j-1})\theta)$. (3)

Base case ($j = 1$): Since $P \cup I \models \forall(\varphi_i)$, we have $P \cup I \models \forall((B_{i,1} \wedge \dots \wedge B_{i,n_i} \rightarrow A_i)\delta_0\theta)$. Hence, if $P \cup I \models \forall((B_{i,1} \wedge \dots \wedge B_{i,n_i})\delta_0\theta)$ then $P \cup I \models \forall(A_i\delta_0\theta)$, which means $P \cup I \models \forall(p(\bar{s}_0)\theta)$.

Induction step: Suppose the induction hypothesis holds for $j \leq n_i$, i.e.,

for every θ , if $P \cup I \models \forall((B_{i,j} \wedge \dots \wedge B_{i,n_i})\delta_{j-1}\theta)$ then $P \cup I \models \forall(p(\bar{s}_{j-1})\theta)$. (4)

We show that it also holds for $j + 1$, i.e.,

for every θ' , if $P \cup I \models \forall((B_{i,j+1} \wedge \dots \wedge B_{i,n_i})\delta_j\theta')$ then $P \cup I \models \forall(p(\bar{s}_j)\theta')$. (5)

Suppose

$P \cup I \models \forall((B_{i,j+1} \wedge \dots \wedge B_{i,n_i})\delta_j\theta')$. (6)

Take $\theta = \gamma_j\theta'$.

- Consider the case $kind(v_j) = extensional$ and let $p_j = pred(v_j)$. By (1), there exist $\bar{t}'_j \in I(p_j)$ and a fresh variant \bar{t}''_j of \bar{t}'_j such that $\gamma_j = mgu(B_{i,j}\delta_{j-1}, p_j(\bar{t}''_j))$, $\bar{s}_j = \bar{s}_{j-1}\gamma_j$ and $\delta_j = (\delta_{j-1}\gamma_j)_{|post_vars(v_j)}$. We have $P \cup I \models \forall(p_j(\bar{t}'_j))$, hence $P \cup I \models \forall(p_j(\bar{t}''_j)\gamma_j)$, which means $P \cup I \models \forall(B_{i,j}\delta_{j-1}\gamma_j)$. Hence $P \cup I \models \forall(B_{i,j}\delta_{j-1}\gamma_j\theta')$, which means

$P \cup I \models \forall(B_{i,j}\delta_{j-1}\theta)$. (7)

Since $\delta_j = (\delta_{j-1}\gamma_j)|_{\text{post_vars}(v_j)}$ and $\theta = \gamma_j\theta'$, we have that

$$(B_{i,j+1} \wedge \dots \wedge B_{i,n_i})\delta_j\theta' = (B_{i,j+1} \wedge \dots \wedge B_{i,n_i})\delta_{j-1}\theta.$$

This together with (6), (7) and (4) implies $P \cup I \models \forall(p(\bar{s}_{j-1})\theta)$. Since $\bar{s}_{j-1}\theta = \bar{s}_{j-1}\gamma_j\theta' = \bar{s}_j\theta'$, it follows that $P \cup I \models \forall(p(\bar{s}_j)\theta')$, which completes the proof of (5) for the case $\text{kind}(v_j) = \text{extensional}$.

- Consider the case $\text{kind}(v_j) = \text{intensional}$ and let $p_j = \text{pred}(v_j)$. By (2), there exist $\bar{t}'_j \in \text{tuples}(\text{ans}_j)$ and a fresh variant \bar{t}''_j of \bar{t}'_j such that $\gamma_j = \text{mgu}(B_{i,j}\delta_{j-1}, p_j(\bar{t}''_j))$, $\bar{s}_j = \bar{s}_{j-1}\gamma_j$ and $\delta_j = (\delta_{j-1}\gamma_j)|_{\text{post_vars}(v_j)}$. By the inductive assumption of the outer induction, we have $P \cup I \models \forall(p_j(\bar{t}''_j))$, hence $P \cup I \models \forall(p_j(\bar{t}''_j)\gamma_j)$, which means $P \cup I \models \forall(B_{i,j}\delta_{j-1}\gamma_j)$. Analogously as for the above case, we can derive that $P \cup I \models \forall(p(\bar{s}_j)\theta')$, which completes the proof of (5) and (3).

By (3), when $j = n_i + 1$ and $\theta = \varepsilon$, we have that $P \cup I \models \forall(p(\bar{s}_{n_i}))$, which means $P \cup I \models \forall(p(\bar{t}))$. \triangleleft

We need the following lemma for the completeness theorem. We assume that the sets of fresh variables used for renaming variables of input program clauses in SLD-refutations and in Algorithm 1 are disjoint.

Lemma 4.3. *After a run of Algorithm 1 (using parameter l) on a query $(P, q(\bar{x}))$ and an extensional instance I , for every intensional predicate r of P , for every $\bar{s} \in \text{tuples}(\text{input}_r)$ and for every SLD-refutation of $P \cup I \cup \{\leftarrow r(\bar{s})\}$ that uses the leftmost selection function, does not contain any goal with term-depth greater than l and has a computed answer θ with the term-depth of $\bar{s}\theta$ not greater than l , there exists $\bar{s}' \in \text{tuples}(\text{ans}_r)$ such that $\bar{s}\theta$ is an instance of a variant of \bar{s}' .*

Proof. We prove this lemma by induction on the length of the mentioned SLD-refutation. Let $\theta_1, \dots, \theta_h$ be the sequence of mgu's used in the refutation. We have that $r(\bar{s})\theta_1 \dots \theta_h = r(\bar{s})\theta$. Suppose that the first step of the refutation of $P \cup I \cup \{\leftarrow r(\bar{s})\}$ uses an input program clause $\varphi'_i = (A'_i \leftarrow B'_{i,1}, \dots, B'_{i,n_i})$, which is a variant of a program clause $\varphi_i = (A_i \leftarrow B_{i,1}, \dots, B_{i,n_i})$ of P , resulting in the resolvent $\leftarrow (B'_{i,1}, \dots, B'_{i,n_i})\theta_1$. Let $k_1 = 2$, $k_{n_i+1} = h + 1$ and suppose that, for $1 \leq j \leq n_i$,

$$\begin{aligned} &\text{the fragment for processing } \leftarrow B'_{i,j}\theta_1 \dots \theta_{k_j-1} \text{ of the refutation of} \\ &P \cup I \cup \{\leftarrow r(\bar{s})\} \text{ uses mgu's } \theta_{k_j}, \dots, \theta_{k_{j+1}-1}. \end{aligned} \quad (8)$$

Thus, after processing the atom $B'_{i,j-1}$ for $2 \leq j \leq n_i + 1$, the next goal of the refutation of $\leftarrow r(\bar{s})$ is $\leftarrow (B'_{i,j}, \dots, B'_{i,n_i})\theta_1 \dots \theta_{k_j-1}$. (If $j = n_i + 1$ then the goal is empty.)

Let ϱ be a renaming substitution such that $\varphi'_i = \varphi_i\varrho$. Thus, $B'_{i,j} = B_{i,j}\varrho$ for $1 \leq j \leq n_i$. We can assume that ϱ does not use any variable occurring in \bar{s} .

We will refer to the data structures used by Algorithm 1.

We first prove the following remark:

Remark 4.4. Let $1 \leq j \leq n_i$, $v = \text{filter}_{i,j}$, $u = \text{filter}_{i,j-1}$ if $j > 1$, and $u = \text{pre-filter}_i$ otherwise. If $(\bar{t}_{j-1}, \delta_{j-1})$ is a subquery transferred through (u, v) at some step and there exists a substitution η such that

$$(A_i, (B_{i,j}, \dots, B_{i,n_i}))\varrho\theta_1 \dots \theta_{k_j-1} = (r(\bar{t}_{j-1}), (B_{i,j}, \dots, B_{i,n_i})\delta_{j-1})\eta \quad (9)$$

then there exist a subquery (\bar{t}_j, δ_j) transferred through $(v, \text{succ}(v))$ at some step and a substitution η' such that

$$(A_i, (B_{i,j+1}, \dots, B_{i,n_i}))\varrho\theta_1 \dots \theta_{k_{j+1}-1} = (r(\bar{t}_j), (B_{i,j+1}, \dots, B_{i,n_i})\delta_j)\eta' \quad (10)$$

Suppose the premises of this remark hold. Without loss of generality we assume that:

$$\begin{aligned} & \text{if } (kind(v) = \textit{extensional} \text{ and } T(v) = \textit{true}) \text{ or } kind(v) = \textit{intensional} \\ & \text{then the subquery } (\bar{t}_{j-1}, \delta_{j-1}) \text{ was added to } \textit{subqueries}(v). \end{aligned} \quad (11)$$

Since $B'_{i,j} = B_{i,j} \varrho$ and (9), we have that:

$$(\leftarrow B'_{i,j} \theta_1 \dots \theta_{k_j-1}) = (\leftarrow B_{i,j} \varrho \theta_1 \dots \theta_{k_j-1}) = (\leftarrow B_{i,j} \delta_{j-1} \eta). \quad (12)$$

Since the term-depth of $B_{i,j} \delta_{j-1} \eta = B'_{i,j} \theta_1 \dots \theta_{k_j-1}$ is not greater than l , the term-depth of $B_{i,j} \delta_{j-1}$ is also not greater than l . By (8), (12) and Lifting Lemma 2.2, we have that

$$\begin{aligned} & \text{there exists a refutation of } P \cup I \cup \{\leftarrow B_{i,j} \delta_{j-1}\} \text{ using the leftmost} \\ & \text{selection function and mgu's } \theta'_{k_j}, \dots, \theta'_{k_{j+1}-1} \text{ such that the term-depths} \\ & \text{of goals are not greater than } l \text{ and } \eta \theta_{k_j} \dots \theta_{k_{j+1}-1} = \theta'_{k_j} \dots \theta'_{k_{j+1}-1} \mu \text{ for} \\ & \text{some substitution } \mu. \end{aligned} \quad (13)$$

Consider the case when the predicate $p = \textit{pred}(v)$ of $B_{i,j}$ is an extensional predicate.

Thus,

$$k_{j+1} = k_j + 1 \quad (14)$$

and

$$B_{i,j} \delta_{j-1} \theta'_{k_j} = p(\bar{t}') \sigma \theta'_{k_j} \quad (15)$$

where $p(\bar{t}') \sigma$ is the input program clause used for resolving $\leftarrow B_{i,j} \delta_{j-1}$, with $\bar{t}' \in I(p)$ and σ being a renaming substitution. Regarding the transfer of the subquery $(\bar{t}_{j-1}, \delta_{j-1})$ through (u, v) , under the assumption (11), Algorithm 1 unifies $\textit{atom}(v) \delta_{j-1} = B_{i,j} \delta_{j-1}$ with a fresh variant $p(\bar{t}') \sigma'$ of $p(\bar{t}')$, where σ' is a renaming substitution, resulting in an mgu γ (by (15), $B_{i,j} \delta_{j-1}$ and $p(\bar{t}') \sigma'$ are unifiable, which is also justified below) and then transfers the subquery $(\bar{t}_{j-1} \gamma, (\delta_{j-1} \gamma)_{| \textit{post_vars}(v)})$ through $(v, \textit{succ}(v))$. Let

$$\bar{t}_j = \bar{t}_{j-1} \gamma \text{ and } \delta_j = (\delta_{j-1} \gamma)_{| \textit{post_vars}(v)}. \quad (16)$$

We have that $\sigma = \sigma' \sigma''$ for some renaming substitution σ'' such that

$$\sigma'' \text{ does not use variables of } \bar{t}_{j-1}, \delta_{j-1} \text{ and } \textit{pre_vars}(v). \quad (17)$$

Thus $B_{i,j} \delta_{j-1} \sigma'' \theta'_{k_j} = B_{i,j} \delta_{j-1} \theta'_{k_j}$, and by (15) and the fact $\sigma = \sigma' \sigma''$, we have that

$$(B_{i,j} \delta_{j-1}) \sigma'' \theta'_{k_j} = B_{i,j} \delta_{j-1} \theta'_{k_j} = p(\bar{t}') \sigma \theta'_{k_j} = (p(\bar{t}') \sigma') \sigma'' \theta'_{k_j}.$$

Hence, $B_{i,j} \delta_{j-1}$ and $p(\bar{t}') \sigma'$ are unifiable using $\sigma'' \theta'_{k_j}$, while γ is an mgu for them. Hence

$$\sigma'' \theta'_{k_j} = \gamma \mu' \quad (18)$$

for some substitution μ' . Let $\eta' = \mu' \mu$. We have that:

$$\begin{aligned} & (A_i, (B_{i,j+1}, \dots, B_{i,n_i})) \varrho \theta_1 \dots \theta_{k_{j+1}-1} \\ &= ((A_i, (B_{i,j+1}, \dots, B_{i,n_i})) \varrho \theta_1 \dots \theta_{k_j-1}) \theta_{k_j} \dots \theta_{k_{j+1}-1} \\ &= (r(\bar{t}_{j-1}), (B_{i,j+1}, \dots, B_{i,n_i}) \delta_{j-1}) \eta \theta_{k_j} \dots \theta_{k_{j+1}-1} \text{ (by the assumption (9))} \\ &= (r(\bar{t}_{j-1}), (B_{i,j+1}, \dots, B_{i,n_i}) \delta_{j-1}) \theta'_{k_j} \dots \theta'_{k_{j+1}-1} \mu \text{ (by (13))} \\ &= (r(\bar{t}_{j-1}), (B_{i,j+1}, \dots, B_{i,n_i}) \delta_{j-1}) \sigma'' \theta'_{k_j} \dots \theta'_{k_{j+1}-1} \mu \text{ (by (17))} \\ &= (r(\bar{t}_{j-1}), (B_{i,j+1}, \dots, B_{i,n_i}) \delta_{j-1}) \gamma \mu' \mu \text{ (by (14) and (18))} \\ &= (r(\bar{t}_j), (B_{i,j+1}, \dots, B_{i,n_i}) \delta_j) \eta' \text{ (by (16) and the fact } \eta' = \mu' \mu). \end{aligned}$$

We have shown (10) and thus proved Remark 4.4 for the case when the predicate of $B_{i,j}$ is extensional.

Now consider the case when the predicate p of $B_{i,j}$ is an intensional predicate.

By the assumption (11), the subquery $(\bar{t}_{j-1}, \delta_{j-1})$ was also added to $unprocessed_subqueries_2(v)$. Let $B_{i,j}\delta_{j-1} = p(\bar{t}')$. There must exist some tuple \bar{t} more general than \bar{t}' that was added to $tuples(input_p)$ at some step. Let $\bar{t}\alpha = \bar{t}'$ for some substitution α that uses only variables from \bar{t} and \bar{t}' . Thus,

$$B_{i,j}\delta_{j-1} = p(\bar{t})\alpha \quad (19)$$

By (13) and Lifting Lemma 2.2, it follows that there exists a refutation of $PUI \cup \{\leftarrow p(\bar{t})\}$ using the leftmost selection function and mgu's $\theta''_{k_j}, \dots, \theta''_{k_{j+1}-1}$ such that the term-depths of the goals are not greater than l and

$$\alpha\theta'_{k_j} \dots \theta'_{k_{j+1}-1} = \theta''_{k_j} \dots \theta''_{k_{j+1}-1}\beta \quad (20)$$

for some substitution β . By the inductive assumption, $tuples(ans_p)$ contains a tuple \bar{t}'' such that $\bar{t}\theta''_{k_j} \dots \theta''_{k_{j+1}-1}$ is an instance of a variant of \bar{t}'' . Since

$$\begin{aligned} B_{i,j}\delta_{j-1}\theta'_{k_j} \dots \theta'_{k_{j+1}-1} &= p(\bar{t})\alpha\theta'_{k_j} \dots \theta'_{k_{j+1}-1} \quad (\text{by (19)}) \\ &= p(\bar{t})\theta''_{k_j} \dots \theta''_{k_{j+1}-1}\beta \quad (\text{by (20)}), \end{aligned}$$

it follows that

$$B_{i,j}\delta_{j-1}\theta'_{k_j} \dots \theta'_{k_{j+1}-1} \text{ is an instance of a variant of } p(\bar{t}''). \quad (21)$$

From certain moment there were both $(\bar{t}_{j-1}, \delta_{j-1}) \in subqueries(v)$ and $\bar{t}'' \in tuples(ans_p)$. Hence, at some step Algorithm 1 unified $atom(v)(\delta_{j-1}) = B_{i,j}\delta_{j-1}$ with a fresh variant $p(\bar{t}'')\sigma$ of $p(\bar{t}'')$, where σ is a renaming substitution. The atom $p(\bar{t}'')\sigma$ does not contain variables of \bar{t}_{j-1} , δ_{j-1} , $pre_vars(v)$ and $\theta'_{k_j} \dots \theta'_{k_{j+1}-1}$. By (21), $B_{i,j}\delta_{j-1}$ and $p(\bar{t}'')\sigma$ are unifiable. Let the resulting mgu be γ and let

$$\bar{t}_j = \bar{t}_{j-1}\gamma \text{ and } \delta_j = (\delta_{j-1}\gamma)|_{post_vars(v)}. \quad (22)$$

Algorithm 1 then transferred the subquery (\bar{t}_j, δ_j) through $(v, succ(v))$.

By (21), $B_{i,j}\delta_{j-1}\theta'_{k_j} \dots \theta'_{k_{j+1}-1}$ is an instance of $p(\bar{t}'')\sigma$. Let ρ be a substitution with domain contained in $Vars(p(\bar{t}'')\sigma)$ such that $B_{i,j}\delta_{j-1}\theta'_{k_j} \dots \theta'_{k_{j+1}-1} = p(\bar{t}'')\sigma\rho$. We have that

$$\begin{aligned} &\text{the domain of } \rho \text{ does not contain variables of } \bar{t}_{j-1}, \delta_{j-1}, pre_vars(v) \text{ and} \\ &\theta'_{k_j} \dots \theta'_{k_{j+1}-1} \end{aligned} \quad (23)$$

and $\theta'_{k_j} \dots \theta'_{k_{j+1}-1} \cup \rho$ is a unifier for $B_{i,j}\delta_{j-1}$ and $p(\bar{t}'')\sigma$. As γ is an mgu for $B_{i,j}\delta_{j-1}$ and $p(\bar{t}'')\sigma$, we have that

$$\gamma\mu' = (\theta'_{k_j} \dots \theta'_{k_{j+1}-1} \cup \rho) \quad (24)$$

for some substitution μ' . Let $\eta' = \mu'\mu$. We have that:

$$\begin{aligned} &(A_i, (B_{i,j+1}, \dots, B_{i,n_i}))\varrho\theta_1 \dots \theta_{k_{j+1}-1} \\ &= (r(\bar{t}_{j-1}), (B_{i,j+1}, \dots, B_{i,n_i})\delta_{j-1})\theta'_{k_j} \dots \theta'_{k_{j+1}-1}\mu \quad (\text{as shown before}) \\ &= (r(\bar{t}_{j-1}), (B_{i,j+1}, \dots, B_{i,n_i})\delta_{j-1})(\theta'_{k_j} \dots \theta'_{k_{j+1}-1} \cup \rho)\mu \quad (\text{by (23)}) \\ &= (r(\bar{t}_{j-1}), (B_{i,j+1}, \dots, B_{i,n_i})\delta_{j-1})\gamma\mu'\mu \quad (\text{by (24)}) \\ &= (r(\bar{t}_j), (B_{i,j+1}, \dots, B_{i,n_i})\delta_j)\eta' \quad (\text{by (22) and the fact } \eta' = \mu'\mu). \end{aligned}$$

We have shown (10) and thus proved Remark 4.4 for the case when the predicate of $B_{i,j}$ is intensional. This completes the proof of this remark.

Recall that $r(\bar{s})\varrho = r(\bar{s})$. Since $\theta_1 = \text{mgu}(r(\bar{s}), A'_i)$ and $A'_i = A_i\varrho$, it follows that $r(\bar{s})\varrho\theta_1 = r(\bar{s})\theta_1 = A'_i\theta_1 = A_i\varrho\theta_1$ and hence $\varrho\theta_1$ is a unifier for $r(\bar{s})$ and A_i . Let γ_0 be the mgu Algorithm 1 uses to unify $r(\bar{s})$ with A_i . Thus, $\gamma_0\eta_0 = \varrho\theta_1$ for some substitution η_0 . Moreover, $(\bar{t}_0, \delta_0) = (\bar{s}\gamma_0, (\gamma_0)_{\text{pre_vars}(\text{filter}_{i,1})})$ is a subquery Algorithm 1 transferred through $(\text{pre_filter}_i, \text{filter}_{i,1})$. Recall that $k_1 = 2$ and observe that the premises of Remark 4.4 hold for $j = 1$ and for the subquery (\bar{t}_0, δ_0) using $\eta = \eta_0$. Hence there exist a subquery (\bar{t}_1, δ_1) transferred through $(\text{filter}_{i,1}, \text{filter}_{i,2})$ at some step and a substitution η_1 such that

$$(A_i, (B_{i,2}, \dots, B_{i,n_i}))\varrho\theta_1 \dots \theta_{k_2-1} = (r(\bar{t}_1), (B_{i,2}, \dots, B_{i,n_i})\delta_1)\eta_1.$$

For each $1 < j \leq n_i$, we can apply Remark 4.4 to obtain a subquery (\bar{t}_j, δ_j) and η_j (for η'). Since $\text{post_vars}(\text{filter}_{i,n_i}) = \emptyset$, it follows that, for $j = n_i$, we have that $(\bar{t}_{n_i}, \varepsilon)$ is a subquery transferred through $(\text{filter}_{i,n_i}, \text{post_filter}_i)$ at some step and

$$A_i\varrho\theta_1 \dots \theta_{k_{n_i+1}-1} = r(\bar{t}_{n_i})\eta_{n_i}.$$

Since $k_{n_i+1} = h + 1$ and $\theta = (\theta_1 \dots \theta_h)_{\text{Vars}(\bar{s})}$, it follows that

$$r(\bar{s})\theta = r(\bar{s})\theta_1 \dots \theta_h = A'_i\theta_1 \dots \theta_h = A_i\varrho\theta_1 \dots \theta_h = r(\bar{t}_{n_i})\eta_{n_i}.$$

Thus, $\bar{s}\theta$ is an instance of \bar{t}_{n_i} . Since $(\bar{t}_{n_i}, \varepsilon)$ was transferred through $(\text{filter}_{i,n_i}, \text{post_filter}_i)$, $\text{tuples}(\text{ans}_r)$ will contain \bar{s}' such that \bar{t}_{n_i} is an instance of a variant of \bar{s}' . Clearly, $\bar{s}\theta$ is also an instance of that variant of \bar{s}' . This completes the proof. \triangleleft

Theorem 4.5 (Completeness). *After a run of Algorithm 1 (using parameter l) on a query $(P, q(\bar{x}))$ and an extensional instance I , for every SLD-refutation of $P \cup I \cup \{\leftarrow q(\bar{x})\}$ that uses the leftmost selection function, does not contain any goal with term-depth greater than l and has a computed answer θ with term-depth not greater than l , there exists $\bar{s} \in \text{tuples}(\text{ans}_q)$ such that $\bar{x}\theta$ is an instance of a variant of \bar{s} .* \triangleleft

This theorem immediately follows from Lemma 4.3. Together with Theorem 2.1 (on completeness of SLD-resolution) it makes a relationship between correct answers of $P \cup I \cup \{\leftarrow q(\bar{x})\}$ and the answers computed by Algorithm 1 for the query $(P, q(\bar{x}))$ on the extensional instance I .

For queries and extensional instances without function symbols, we take term-depth bound $l = 0$ and obtain the following completeness result, which immediately follows from the above theorem.

Corollary 4.6. *After a run of Algorithm 1 using $l = 0$ on a query $(P, q(\bar{x}))$ and an extensional instance I that do not contain function symbols, for every computed answer θ of an SLD-refutation of $P \cup I \cup \{\leftarrow q(\bar{x})\}$ that uses the leftmost selection function, there exists $\bar{t} \in \text{tuples}(\text{ans}_q)$ such that $\bar{x}\theta$ is an instance of a variant of \bar{t} .* \triangleleft

5 Data Complexity

In this subsection we estimate the *data complexity* of Algorithm 1, which is measured w.r.t. the size of the extensional instance I when the query $(P, q(\bar{x}))$ and the term-depth bound l are fixed.

If terms are represented as sequences of symbols or as trees then there will be a problem with complexity. Namely, unifying the terms $f(x_1, \dots, x_n)$ and

$f(g(x_0, x_0), \dots, g(x_{n-1}, x_{n-1}))$, we get a term of exponential length.⁴ If the term-depth bound l is used in all steps, including the ones of unification, then the problem will not arise. But we do not want to be so restrictive.

To represent a term we use instead a rooted acyclic directed graph which is permitted to have multiple ordered arcs and caches nodes representing the same subterm. Such a graph will simply be called a DAG. As an example, the DAG of $f(x, a, x)$ has the root n_f labeled by f , a node n_x labeled by x , a node n_a labeled by a , and three ordered edges outgoing from n_f : the first one and the third one are connected to n_x , while the second one is connected to n_a .

The *size of a term* t , denoted by $size(t)$, is defined to be the size of the DAG of t (i.e. the number of nodes and edges of the DAG of t). The sizes of other term-based expressions or data structures are defined as usual. For example, we define:

- the *size of a tuple* (t_1, \dots, t_k) to be $size(t_1) + \dots + size(t_k)$
- the *size of a set of tuples* to be the sum of the sizes of those tuples
- the *size of a substitution* $\{x_1/t_1, \dots, x_k/t_k\}$ to be $k + size(t_1) + \dots + size(t_k)$
- the *size of a node* v of a *QSQ-net* (V, E, T, C) to be the sum of the sizes of the components of $C(v)$.

Using DAGs to represent terms, unification of two atoms A and A' can be done in polynomial time in the sizes of A and A' . In the case A and A' are unifiable, the resulting atom and the resulting mgu have sizes that are polynomial in the sizes of A and A' . Similarly, checking whether A is an instance of A' can also be done in polynomial time in the sizes of A and A' .

The following theorem estimates the data complexity of Algorithm 1, under the assumption that terms are represented by DAGs and unification and checking instances of atoms are done in polynomial time.

Theorem 5.1. *For a fixed query and a fixed bound l on term-depth, Algorithm 1 runs in polynomial time in the size of the extensional instance.*

Proof. Consider a run of Algorithm 1 using parameter l on a query $(P, q(\bar{x}))$ and on an extensional instance I with size n . Here, $(P, q(\bar{x}))$ and l are fixed. Thus, for every $1 \leq i \leq m$, n_i is bounded by a constant. Similarly, if p is an intensional predicate from P then the arity of p is also bounded by a constant.

Observe that the number of tuples that are added to any set of the form $tuples(input_p)$ or $tuples(ans_p)$ are bounded by a polynomial of n . The reasons are:

- intensional predicates come from P
- constant symbols and function symbols come from P and I
- $tuples(input_p)$ and $tuples(ans_p)$ consist of tuples with term-depth bounded by l
- a tuple is added to a set of the form $tuples(input_p)$ or $tuples(ans_p)$ only when it is not an instance of a fresh variant of any tuple from the set
- a tuple is deleted from a set of the form $tuples(input_p)$ or $tuples(ans_p)$ only when it is an instance of a new tuple added to the set.

For similar reasons, the number of subqueries that are added to any set of the form $subqueries(v)$ are also bounded by a polynomial of n .

Consequently, the sizes of sets of the form $tuples(input_p)$, $tuples(ans_p)$, $subqueries(v)$, $unprocessed(v, w)$, $unprocessed_tuples(v)$, $unprocessed_subqueries(v)$ or $unprocessed_subqueries_2(v)$ are bounded by a polynomial of n . Therefore, the size of the

⁴ Another example is the pair $f(x_1, \dots, x_n, x_1, \dots, x_n)$ and $f(y_1, \dots, y_n, g(y_0, y_0), \dots, g(y_{n-1}, y_{n-1}))$.

constructed QSQ-net is bounded by a polynomial of n , and any execution of procedure **transfer**, procedure **fire** or function **active-edge** is done in polynomial time in n .

A transfer or a firing for an edge (u, v) is done only when a new tuple was added to $tuples(u)$ or a new subquery was added to $subqueries(u)$. Thus, we can conclude that Algorithm 1 runs in polynomial time in n . \triangleleft

Corollary 5.2. *Algorithm 1 with term-depth bound $l = 0$ is a complete evaluation algorithm with PTIME data complexity for the class of queries over a signature without function symbols.* \triangleleft

This corollary follows from Theorem 4.2 (on soundness), Corollary 4.6 (on completeness) and the above theorem (on data complexity).

6 QSQ-Nets with Tail Recursion Elimination

A query-subquery net structure with tail recursion elimination (in short, *QSQTRE-net structure*) of P is a tuple (V, E, T) defined similarly to a QSQ-net structure of P , but with the following differences:

- T is a function, called the *type* of the net structure, mapping
 - each $filter_{i,j} \in V$ such that the predicate of $B_{i,j}$ is extensional to *true* or *false*
 - each intensional predicate to *true* or *false*.
- If A_i and B_{i,n_i} have the same intensional predicate p with $T(p) = \text{true}$ then V does not contain the node $post_filter_i$ and E does not contain the edges $(filter_{i,n_i}, post_filter_i)$, $(post_filter_i, ans_p)$ and $(ans_p, filter_{i,n_i})$.

The function T can thus be called a memorizing type for extensional nodes $filter_{i,j}$ (as in QSQ-net structures), and a tail-recursion-elimination type for intensional predicates.

Example 6.1. Reconsider the positive logic program given in Example 3.1:

$$\begin{aligned} p(x, y) &\leftarrow q(x, y) \\ p(x, y) &\leftarrow q(x, z), p(z, y). \end{aligned}$$

A QSQTRE-net structure (V, E, T) of this program with $T(p) = \text{true}$ has the topological structure illustrated in Figure 2, which is like a loop. \triangleleft

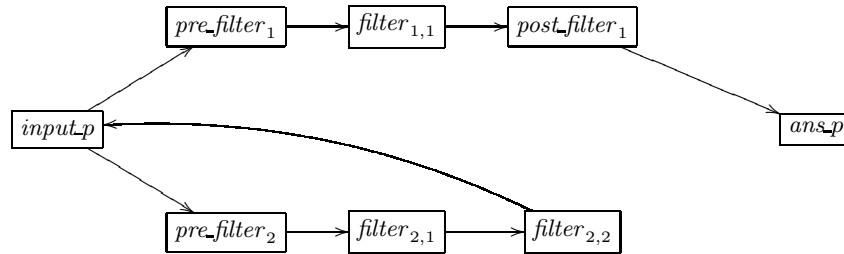


Fig. 2. An illustration for Example 6.1.

A query-subquery net with tail recursion elimination (in short, *QSQTRE-net*) of P is a tuple $N = (V, E, T, C)$ defined similarly to a QSQ-net of P , but with the following differences:

- (V, E, T) is a QSQTRE-net structure of P
- if $v = \text{input}_p$ and $T(p) = \text{true}$ then $C(v)$ consists of:
 - $\text{tuple_pairs}(v)$: a set of pairs of generalized tuples of the same arity as p
 - $\text{unprocessed}(v, w)$ for $(v, w) \in E$: a subset of $\text{tuple_pairs}(v)$
- if $v = \text{filter}_{i, n_i}$, $\text{kind}(v) = \text{intensional}$, $\text{pred}(v) = p$ and $T(p) = \text{true}$ then the structure $C(v)$ does not contain $\text{unprocessed_subqueries}(v)$ and $\text{unprocessed_tuples}(v)$.

The intuition behinds a pair $(\bar{t}, \bar{t}') \in \text{tuple_pairs}(\text{input}_p)$ is that:

- \bar{t} is a usual input tuple for p , but the intended goal at a higher level is $\leftarrow p(\bar{t}')$
- any correct answer for $P \cup I \cup \{\leftarrow p(\bar{t})\}$ is also a correct answer for $P \cup I \cup \{\leftarrow p(\bar{t}')\}$
- if a substitution θ is a computed answer of $P \cup I \cup \{\leftarrow p(\bar{t})\}$ then we will store in ans_p the tuple $\bar{t}'\theta$ instead of $\bar{t}\theta$.

Data transferred through an edge of the form (input_p, v) or (v, input_p) in a QSQTRE-net (V, E, T, C) , where p is an intensional predicate with $T(p) = \text{true}$, is redefined to be a finite set of pairs of generalized tuples of the same arity as p .

We say that a tuple pair (\bar{t}, \bar{t}') is *more general* than (\bar{t}_2, \bar{t}_2') , and (\bar{t}_2, \bar{t}_2') is an *instance* of (\bar{t}, \bar{t}') , if there exists a substitution θ such that $(\bar{t}, \bar{t}')\theta = (\bar{t}_2, \bar{t}_2')$.

Other notions for QSQTRE-nets are defined similarly as for QSQ-nets.

Procedure **transfer2**(D, u, v) (given on page 21) is a modified version of **transfer**(D, u, v) for dealing with tail recursion elimination.

Let procedure **fire2**(u, v) be the modified version of **fire**(u, v) obtained by:

- changing the calls of **transfer** by calls of **transfer2** (with the same parameters)
- replacing Step 15 by macro **compute-gamma** defined on page 21.

Algorithm 2 (given on page 22) is our reformulation of Algorithm 1 by using QSQTRE-nets for evaluating queries.

Theorem 6.2. *Theorems 4.2, 4.5, 5.1 and Corollaries 4.6, 5.2 still hold when “Algorithm 1” is replaced by “Algorithm 2”.* \triangleleft

7 Control Strategies

Recall that in Algorithms 1 and 2 we repeatedly select an active edge and fire the operation for it. Such selection is decided by the adopted control strategy, which can be arbitrary. In this section we describe two control strategies: the first one is to reduce the number of accesses to the secondary storage, while the second one is depth-first search, which gives priority to the order of clauses in the positive logic program defining intensional predicates and thus allows the user to control the evaluation to a certain extent.

7.1 Reducing the Number of Accesses to the Secondary Storage

It is reasonable to assume that the computer memory is not large enough to load the whole extensional instance of the knowledge base into it and evaluation of queries cannot usually be done totally in the computer memory. Note that, not only extensional relations may be too large, but temporary relations used for computing intensional predicates like $\text{tuples}(v)$, $\text{unprocessed}(v, w)$, $\text{subqueries}(v)$, ... may also be too large. Therefore, sometimes we have to load a relation into the computer memory, and sometimes we have to unload a relation to the secondary storage. As access to the secondary storage is time-consuming, it is desirable to reduce the total number of such accesses. Here is a strategy for this:

Procedure transfer2(D, u, v)

Global data: a Horn knowledge base (P, I) , a QSQTRE-net $N = (V, E, T, C)$ of P , and a term-depth bound l .

Input: data D to transfer through the edge $(u, v) \in E$.

```

1 if  $D = \emptyset$  then return;
2 if  $u$  is input- $p$  and  $T(p) = \text{true}$  then
3    $\Gamma := \emptyset$ ;
4   foreach  $(\bar{t}, \bar{t}') \in D$  do
5     if  $p(\bar{t})$  and  $\text{atom}(v)$  are unifiable by an mgu  $\gamma$  then
6        $\text{add-subquery}(\bar{t}'\gamma, \gamma|_{\text{post\_vars}(v)}, \Gamma, \text{succ}(v))$ 
7    $\text{transfer}(\Gamma, v, \text{succ}(v))$ 
8 else if  $v$  is input- $p$  and  $T(p) = \text{true}$  then
9   foreach  $(\bar{t}, \bar{t}') \in D$  do
10    let  $(\bar{t}_2, \bar{t}'_2)$  be a fresh variant of  $(\bar{t}, \bar{t}')$ ;
11    if  $(\bar{t}_2, \bar{t}'_2)$  is not an instance of any pair from  $\text{tuple\_pairs}(v)$  then
12      foreach  $(\bar{t}_3, \bar{t}'_3) \in \text{tuple\_pairs}(v)$  do
13        if  $(\bar{t}_3, \bar{t}'_3)$  is an instance of  $(\bar{t}_2, \bar{t}'_2)$  then
14          delete  $(\bar{t}_3, \bar{t}'_3)$  from  $\text{tuple\_pairs}(v)$ ;
15          foreach  $(v, w) \in E$  do delete  $(\bar{t}_3, \bar{t}'_3)$  from  $\text{unprocessed}(v, w)$ 
16        add  $(\bar{t}_2, \bar{t}'_2)$  to  $\text{tuple\_pairs}(v)$ ;
17        foreach  $(v, w) \in E$  do add  $(\bar{t}_2, \bar{t}'_2)$  to  $\text{unprocessed}(v, w)$ 
18 else if  $v$  is filter- $i, n_i$ ,  $\text{kind}(v) = \text{intensional}$ ,  $\text{pred}(v) = p$  and  $T(p) = \text{true}$  then
19   foreach  $(\bar{t}, \delta) \in D$  do
20     if  $\text{term-depth}(\text{atom}(v)\delta) \leq l$  then
21       if no subquery in  $\text{subqueries}(v)$  is more general than  $(\bar{t}, \delta)$  then
22         delete from  $\text{subqueries}(v)$  all subqueries less general than  $(\bar{t}, \delta)$ ;
23         delete from  $\text{unprocessed\_subqueries}_2(v)$  all subqueries less general than  $(\bar{t}, \delta)$ ;
24         add  $(\bar{t}, \delta)$  to both  $\text{subqueries}(v)$  and  $\text{unprocessed\_subqueries}_2(v)$ 
25 else
26   Steps 2-43 of procedure transfer (given on page 10) with the recursive calls of transfer
    replaced by calls of transfer2

```

Procedure add-tuple-pair($\bar{t}, \bar{t}', \Gamma$)

Purpose: add the pair of tuples (\bar{t}, \bar{t}') to Γ , but keep in Γ only the most general pairs.

```

1 let  $(\bar{t}_2, \bar{t}'_2)$  be a fresh variant of  $(\bar{t}, \bar{t}')$ ;
2 if  $(\bar{t}_2, \bar{t}'_2)$  is not an instance of any pair from  $\Gamma$  then
3   delete from  $\Gamma$  all pairs that are instances of  $(\bar{t}_2, \bar{t}'_2)$ ;
4   add  $(\bar{t}_2, \bar{t}'_2)$  to  $\Gamma$ 

```

Procedure compute-gamma

Purpose: a macro used in procedure **fire2** for replacing Step 15 of procedure **fire**.

```

1 if  $T(p) = \text{false}$  then
2   foreach  $(\bar{t}, \delta) \in \text{unprocessed\_subqueries}_2(u)$  do let  $p(\bar{t}') = \text{atom}(u)\delta$ ,  $\text{add-tuple}(\bar{t}', \Gamma)$ 
3 else if  $j < n_i$  then
4   foreach  $(\bar{t}, \delta) \in \text{unprocessed\_subqueries}_2(u)$  do let  $p(\bar{t}') = \text{atom}(u)\delta$ ,  $\text{add-tuple-pair}(\bar{t}', \bar{t}', \Gamma)$ 
5 else
6   foreach  $(\bar{t}, \delta) \in \text{unprocessed\_subqueries}_2(u)$  do let  $p(\bar{t}') = \text{atom}(u)\delta$ ,  $\text{add-tuple-pair}(\bar{t}', \bar{t}, \Gamma)$ 

```

Algorithm 2: for evaluating a query $(P, q(\bar{x}))$ on an extensional instance I .

```

let  $(V, E, T)$  be a QSQTRE-net structure of  $P$ ; //  $T$  can be chosen arbitrarily
set  $C$  so that  $N = (V, E, T, C)$  is an empty QSQTRE-net of  $P$ ;
let  $\bar{x}'$  be a fresh variant of  $\bar{x}$ ;
if  $T(q) = \text{false}$  then
     $\text{tuples}(\text{input\_}q) := \{\bar{x}'\}$ ;
    foreach  $(\text{input\_}q, v) \in E$  do  $\text{unprocessed}(\text{input\_}q, v) := \{\bar{x}'\}$ 
else
     $\text{tuple\_pairs}(\text{input\_}q) := \{(\bar{x}', \bar{x}')\}$ ;
    foreach  $(\text{input\_}q, v) \in E$  do  $\text{unprocessed}(\text{input\_}q, v) := \{(\bar{x}', \bar{x}')\}$ 
while there exists  $(u, v) \in E$  such that  $\text{active\_edge}(u, v)$  holds do
    select  $(u, v) \in E$  such that  $\text{active\_edge}(u, v)$  holds;
    // any strategy is acceptable for the above selection
     $\text{fire2}(u, v)$ 
return  $\text{tuples}(\text{ans\_}q)$ 

```

- If (u, v) and (u', v') are active edges of the considered QSQ-net/QSQTRE-net and firing the edge (u, v) can be done in the computer memory, while firing the edge (u', v') requires loading some relations from the secondary storage then the edge (u, v) has a higher priority than (u', v') (for being selected).
- If firing any of edges (u, v) and (u', v') can be done in the computer memory then:
 - the one that could enable a next operation be done in the computer memory (e.g. firing some edge (v, w) or (v', w')) is considered to have a higher priority than the other
 - if both of the edges are equal w.r.t. the above criterion then the one that could enable more next operations be done in the computer memory is considered to have a higher priority than the other
 - if both of the edges are equal w.r.t. the above criteria then the one that processes more tuples/subqueries is considered to have a higher priority than the other.
- If no more operations can be done in the computer memory without loading relations from the secondary storage then select and load such a relation. The criteria for such selection are similar to the above mentioned ones. That is, we choose a relation to load into the computer memory that would enable more next operations be done in the computer memory and that would process more tuples/subqueries.
- If we want to load a relation into the computer memory but there is not enough available space in it then we have to select and unload an in-memory relation to the secondary storage. We can choose the in-memory relation that has not been used in the longest period to unload.

7.2 Depth-First Evaluation

The user may use Prolog programming style to specify the positive logic program defining intensional predicates. In such cases, e.g. as in Example 1.1, the order of the program clauses may be essential and depth-first search may increase efficiency of query evaluation.

For each node of the considered QSQ-net/QSQTRE-net we maintain and update its modification timestamp. For the depth-first evaluation approach, nodes are considered in the decreasing order of modification timestamps. When a node v is considered, we choose an active edge (v, w) to fire. If there is no such an edge, a next node in the mentioned order is chosen for consideration. If there are more than one successor w of v such that the edge (v, w) is active, choose an edge (v, w) according to the following strategy:

- If $v = \text{input}_p$ then w is the node pre_filter_i with the smallest index i such that (v, w) is active (i.e. we consider the program clause φ_i with the smallest index i such that the edge $(v, \text{pre_filter}_i)$ is active).
- If $v = \text{filter}_{i,j}$, $\text{kind}(v) = \text{intensional}$, v has two successors and both the edges $(v, \text{succ}(v))$ and $(v, \text{succ}_2(v))$ are active, then $w = \text{succ}(v)$.⁵
- If $v = \text{ans}_p$ then w is the successor of v with the biggest modification timestamp such that (v, w) is active.

8 Conclusions

We have provided the first framework for developing algorithms for evaluating queries to Horn knowledge bases with the properties that: the approach is goal-directed; each subquery is processed only once and each supplement tuple, if desired⁶, is transferred only once; operations are done set-at-a-time; and any control strategy can be used. The framework forms a generic evaluation method called QSQN. We have proved soundness and completeness of our generic evaluation method and showed that, when the term-depth bound is fixed, the method has PTIME data complexity.

This work is a continuation of [7]. It makes essential improvements: while the QSQR evaluation method of [7] uses iterative deepening search and does redundant recomputations, the QSQN evaluation method developed in this paper allows any control strategy and does not do redundant recomputations. The QSQN evaluation method is much more flexible, e.g., for reducing the number of accesses to the secondary storage.

Our framework is an adaptation and a generalization of the QSQ approach of Datalog for Horn knowledge bases. One of the key differences is that we do not use adornments and annotations, but use substitutions instead. This is natural for the case with function symbols and without the range-restrictedness condition. When restricting to Datalog queries, it groups operations on the same relation together regardless of adornments and allows to reduce the number of accesses to the secondary storage although “joins” and “projections” would be more complicated. QSQ-nets are a more intuitive representation than the description of the QSQ approach of Datalog given in [1]. Our notion of QSQ-net makes a connection to flow networks and is intuitive for developing efficient evaluation algorithms. For example, as shown in the paper, it is easy to incorporate tail recursion elimination into QSQ-nets, and as a result we have QSQTRE-nets.

In comparison with the most well-known evaluation methods, our QSQN evaluation method is more efficient than the QSQR evaluation method (as it does not do redundant recomputations) and is more flexible and thus has essential advantages over the bottom-up evaluation method based on magic-set transformation and improved seminaive evaluation (as shown in Example 1.1).

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Wesley, 1995.
2. K.R. Apt. *From Logic Programming to Prolog*. Prentice-Hall, 1997.
3. F. Bancilhon, D. Maier, Y. Sagiv, and J.D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of PODS'1986*, pages 1–15. ACM, 1986.
4. K.L. Clark. Predicate logic as a computational formalism. Research Report DOC 79/59, Department of Computing, Imperial College, 1979.
5. J. Freire, T. Swift, and D.S. Warren. Taking I/O seriously: Resolution reconsidered for disk. In L. Naish, editor, *Proc. of ICLP'1997*, pages 198–212. MIT Press, 1997.

⁵ If the net is with tail recursion elimination then v may have only one successor.

⁶ when $T(v) = \text{true}$ for all nodes v of the form $\text{filter}_{i,j}$ with $\text{kind}(v) = \text{extensional}$

6. J.W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.
7. E. Madalińska-Bugaj and L.A. Nguyen. A generalized QSQR evaluation method for Horn knowledge bases. To appear in *ACM Transactions on Computational Logic* (available at <http://toc1.acm.org/accepted/455nguyen.pdf>), 2012.
8. W. Nejdl. Recursive strategies for answering recursive queries - the RQA/FQI strategy. In P.M. Stocker, W. Kent, and P. Hammersley, editors, *Proceedings of VLDB'87*, pages 43–50. Morgan Kaufmann, 1987.
9. R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Efficient bottom-up evaluation of logic programs. In J. Vandewalle, editor, *The State of the Art in Computer Systems and Software Engineering*. Kluwer Academic Publishers, 1992.
10. J. Rohmer, R. Lescouer, and J.-M. Kerisit. The Alexander method – a technique for the processing of recursive axioms in deductive databases. *New Generation Computing*, 4(3):273–285, 1986.
11. K.A. Ross. Tail recursion elimination in deductive databases. *ACM Trans. Database Syst.*, 21(2):208–237, 1996.
12. K.F. Sagonas and T. Swift. An abstract machine for tabled execution of fixed-order stratified logic programs. *ACM Trans. Program. Lang. Syst.*, 20(3):586–634, 1998.
13. K.F. Sagonas, T. Swift, and D.S. Warren. XSB as an efficient deductive database engine. In R.T. Snodgrass and M. Winslett, editors, *Proceedings of the 1994 ACM SIGMOD Conference on Management of Data*, pages 442–453. ACM Press, 1994.
14. Y.-D. Shen, L.-Y. Yuan, J.-H. You, and N.-F. Zhou. Linear tabulated resolution based on Prolog control strategy. *TPLP*, 1(1):71–103, 2001.
15. S. Staab. Completeness of the SLD-resolution. Slides of a course on Advanced Data Modeling, <http://www.uni-koblenz.de/FB4/Institutes/IFI/AGStaab/Teaching/SS08/adm08/DB2-SS08-Slides9.ppt>, 2008.
16. R.F. Stärk. A direct proof for the completeness of SLD-resolution. In E. Börger, H.K. Büning, and M.M. Richter, editors, *Proceedings of CSL'89*, volume 440 of *LNCS*, pages 382–383. Springer, 1990.
17. H. Tamaki and T. Sato. OLD resolution with tabulation. In E.Y. Shapiro, editor, *Proceedings of ICLP'1986, LNCS 225*, pages 84–98. Springer, 1986.
18. L. Vieille. Recursive axioms in deductive databases: The query/subquery approach. In *Proceedings of Expert Database Conf.*, pages 253–267, 1986.
19. L. Vieille. A database-complete proof procedure based on SLD-resolution. In *Proceedings of ICLP*, pages 74–103, 1987.
20. L. Vieille. Recursive query processing: The power of logic. *Theor. Comput. Sci.*, 69(1):1–53, 1989.
21. N.-F. Zhou and T. Sato. Efficient fixpoint computation in linear tabling. In *Proceedings of PPDP'2003*, pages 275–283. ACM, 2003.